Science of Computer Programming 21 (1993) 225–261
Elsevier

Axiomatizing CCS, nets and processes *

Nicoletta De Francesco

Università di Pisa, Dipartimento di Ingegneria dell'Informazione, Via Diotisalvi 2, 56100 Pisa, Italy

Ugo Montanari

Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56100 Pisa, Italy

Daniel Yankelevich

Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56100 Pisa, Italy; and HP Labs, Pisa Science Center, Pisa, Italy

Communicated by M. Sintzoff
Received July 1991
Revised November 1992

Abstract

De Francesco, N., U. Montanari and D. Yankelevich, Axiomatizing CCS, nets and processes, Science Computer Programming 21 (1993) 225–261.

Process description languages (PDLs) are appealing for specifying distributed systems mainly because of their compositionality and expressiveness properties. On the other hand, Petri nets, while lacking linguistical structure, offer the advantage of a truly concurrent framework.

Our proposal unifies the models of PDLs and Petri nets in a common specification framework. The binding is given by means of an algebraic approach, which allows us to have in the same algebra both the interleaving and the true concurrency aspects of a distributed system. Starting with the description of a system as a CCS agent, we obtain automatically, by means of axioms, its representation as a Petri net.

The idea of the step by step inclusion of axioms is new in this context, and allows us to construct a complex model by assembling simple pieces.

Moreover, we examine some important aspects of the design of distributed systems, and show how our approach can be useful for dealing with them.

Correspondence to: N. De Francesco, Università di Pisa, Dipartimento di Ingegneria dell'Informazione, Via Diotisalvi 2, 56100 Pisa, Italy. E-mail: nico@iet.unipi.it.

*Research supported in part by Progetto Finalizzato, Sistemi Informatici e Calcolo Parallelo obiettivo LAMBRUSCO, and ESPRIT Basic Research Action 3011, CEDYSIS.

1. Introduction

One of the principal characteristics of distributed systems is the high degree of concurrency among system components. This characteristic discloses a set of problems different from those of traditional concurrent programming, for example concerning operating systems or distributed databases.

Some of the main issues in the specification of distributed systems are concerned with the locality of components, the modular decomposition of a system into independent components, the granularity of the desired parallelism, and the dynamic analysis of the behavior.

A specification language for distributed systems has to take into account these problems: it must be sufficiently powerful to describe all the aspects of a system and at the same time it must present a rather simple and clean semantics to allow the designer to reason about the properties of the various components.

The process description language (PDL) approach [2,3,25,33] is based on an *interleaving semantics*, where the concurrency between events means that they may occur in any order; thus concurrency is reduced, via interleaving, to nondeterminism. In this approach, the operator for the parallel composition is not primitive: given any finite term containing this operator, another term always exists without it, which exhibits the same behavior.

On the other hand, in the *true concurrency* approach (see for instance several papers in [9,24]), the state of a concurrent system is not represented as a monolithic entity. Instead, the system is seen as composed of a set of processes that can proceed independently from each other. Moreover the computations are modeled as partial orders, in which concurrent events are not related and the parallel composition operator is primitive. In this framework, information about *distribution in space* and *causal dependency* are naturally present in the model. Perhaps the most widely used true concurrent model are Petri nets [40], which is the basis of a substantial number of specification languages and methodologies [18,37,41].

We think that a formal specification, while being sufficiently abstract in order to avoid the description of useless implementation details, must also support some design aspects. Thus a conceptual mapping must be provided from the specification to an "abstract" architecture of the system being modeled. In other words, the specification must capture certain implementation relationships between the different parts of the system, and this information must be a guideline to the designer. We think that this need is particularly strong when dealing with distributed systems, where one of the goals is to capture the intrinsic structure of the problem to be solved by means of the program structure itself, i.e. the different processes together with their interactions. If we adopt this point of view, the true concurrency approach is more adequate. The interleaving semantics can be seen as an abstraction of the true concurrency

one, obtained by forgetting spatial and causal dependencies between events [12].

However, PDLs provide some useful advantages for the practical specification of systems. In fact they allow to describe a concurrent system in a compositional way, by specifying separately the various subparts and composing them to produce a new system, which in turn can itself be regarded as a component of another system at a higher level. This modularity property is particularly important when specifying a distributed system, which consists of several components working rather independently and communicating between them. Moreover PDLs can be seen as usual programming languages with advantages in terms of notation conciseness, program structuring, and abstraction.

Instead, no language exists to express Petri nets at least in their classical version: actually a Petri net has to be described by exhaustively presenting its places and transitions without the help of operations to structure and compose its subparts. This lack of compositionality is a well-known drawback of Petri nets.

As a consequence of the above considerations, many approaches have been developed with the aim of capturing the advantages of both PDLs and Petri nets, in terms of modularity and structure from one side and true concurrency from the other. Some of these proposals extend the basic model of Petri nets by enriching it with some information to make composition possible [28,38]. Other approaches, among which ours can be included, start from a process description language and give it a truly concurrent semantics.

The problem of giving a truly concurrent *operational* semantics to CCS in terms of Petri nets has been largely studied in the literature (see, for example, [6,10,11,20,22,23,34,36,42,43]).

Our proposal unifies the models of PDLs and Petri nets in a common specification framework, where the binding is given by means of an algebraic approach, which allows us to have in the same algebra both the interleaving and the true concurrency aspects of a distributed system. Starting with the description of the system by means of CCS, we obtain automatically, by means of axioms, its representation as a Petri net. The constructions presented in this paper have never been all assembled inside the same formalism, in the same algebra, using one notation, and allowing to study their interactions in a simple way. The idea of the step by step inclusion of axioms is also new in this context, and allows us to construct a complex model by assembling simple pieces.

Actually, we restrict the application of the nondeterministic operator of CCS in order to forbid distributed choices. The treatment of full CCS, which has some technical subtleties, can be found in [44].

The main result of this paper is a common framework to describe different aspects of the language. An algebra will be constructed in a modular way,

which contains:

- the transition system (interleaving operational semantics),
- the interleaving computations of the transition system,
- the Petri net (truly concurrent operational semantics),
- the marking graph of the net,
- the nonsequential processes of the net.

All these structures live together in a unique algebra, and one can choose which part of the structure one wants to see, just by using a typing relation defined over the structure. The model is consistent, and the different structures do not interfere. The modular definition is achieved by including axioms which correspond to the definition of each level. As a last level, a set of axioms is included, which establishes the relation between interleaving and truly concurrent semantics. Some consequences of this relation are shown. In this work we do not present formal proofs, in order to simplify the exposition. The proofs of the correctness of the constructions are similar to those that can be found in the referenced papers for each step.

Notice that the construction of the Petri net is done once and for all. This means that we do not give an algorithm to obtain a net from a program, as, for example, in [20,42], but we define one net for the whole language. A similar approach was presented by Milner [32] to introduce the transition system for CCS.

While Petri nets are usually specified by enumerating their places and transitions, in our model nets can be generated by agents. The notation is borrowed from process description languages, which have been widely used for the specification of concurrent systems. At the same time, Petri nets do not lose some attractive properties, as for instance the possibility of graphical manipulation.

We remark that our goal is not to give a new specification language based on Petri nets, but a common basis on which to build several possible truly concurrent specification languages. For example, we do not address the problem of data and consider only the "concurrent part" of the specification model. Nevertheless, we think that the model we have chosen is suitable to be enriched with the specification of data, because of its algebraic nature [15].

In modular and distributed systems a module encapsulates its implementation. It may be useful to be allowed to choose if a module must be a sequential process or a system of cooperating subprocesses, determining in this way the granularity of parallelism. Moreover, it may be useful that the choice between a parallel or a sequential implementation can be done without modifying any of the modules. This facility may be essential when the number of processes becomes very large. This is the case for example in object-oriented distributed systems, where each module is an active object. Various solutions have been proposed. The `domain` construct of Hybrid [35] is a collection of objects which

is seen as a single sequential process, while concurrency occurs only between domains. Also the Presto environment [16] offers this feature.

In order to take this requirement into account, we extend CCS with a new encapsulation construct, whose meaning is that, given a possibly parallel process p, $\{\!|p|\!\}$ has to be seen as a sequential process and thus its parallelism must be expressed by interleaving its events.

One of the goals of the paper is to show how our approach may be useful in the design of distributed systems: some design scenarios are presented and the potential of our approach to face each of them is discussed.

Section 2 addresses some problems occurring in the design of distributed systems. Section 3 recalls some definitions and results from the literature. Section 4 presents our model, whereas Section 5 discusses the usefulness of the approach.

Section 6 makes some considerations about future work.

2. Some design scenarios for distributed systems

In the following we shall describe some problems, arising in the design of distributed systems, which our approach can help to better understand and solve.

Locality

When the number of processes is large, it is important for efficiency reasons to know the "space of action" of a process in order to locate the processes more frequently communicating with it in its neighbourhood. Thus it is important to know, either statically or dynamically, between which processes the communications do occur.

Modularity

Modularity is one of the more important issues in the design of distributed systems, because of the variety of components to be implemented, tested, debugged, and recovered. The following points are concerned with modularity:

- *Recoverability and security.* When a system is composed of a large number of parallel components, the problem of recoverability in the presence of faults is not simple. This task may be aided by the knowledge of the causal dependencies between system components. In fact, if we know which subparts of the system are affected by the faulty one, we can focus on them for recovery. Moreover, knowledge of the causal dependencies makes it simpler to assure the security of the different modules of the system.

- *Compositionality.* When specifying large real systems, it is essential to provide the designer with the ability of composing a system from its parts in a modular way. Thus, the specification language should have compositional properties.
- *Dynamic analysis.* This task is concerned with detecting errors during program execution. Once an error has occurred, it is necessary to locate its cause. If the model describes a computation by a large number of different states, it may be critical to examine all of them. The number of states is very large with an interleaving semantics, where states are "global". If, on the contrary, states are partitioned into distributed subparts as in a Petri net, the dynamic behavior of the system can be represented in a more compact way and this makes it easier to identify the errors.
- *Configuration.* When the problem itself has a distributed structure and there is no central locus of control, it may be useful to maintain the information on the spatial distribution of its components. In the case of distributed systems specification, this information can be reflected by the syntactic structure of the program, for example recording at what nesting level a process is placed with respect to the processes cooperating with it.

Encapsulation

The problem of encapsulation of parallelism, closely related to locality and modularity, concerns the decision about the granularity of parallelism inside a concurrent program. In fact in many cases, in order not to excessively increase the number of concurrent components, it may be useful to hide the parallelism occurring within a certain component and to consider it as a sequential process. This is only possible if:

(1) the distinction between parallelism and sequentiality is clear at the specification level; and
(2) the two concepts are expressible in the same specification language, in order to give the freedom to choose an interleaving or true concurrency implementation for each part of the system.

3. Background

3.1. Place/transition nets

A *place/transition net* [4,40] consists of two sets S and T of places and transitions respectively. To every transition t, two (usually finite) multisets of places ${}^\bullet t$ and $t^\bullet$, called *preset* and *postset*, are associated. Global states consist of *markings*, i.e. of multisets of places. Given a marking u, a transition t is enabled by it, if ${}^\bullet t \subseteq u$, and in this case the *step* $u\ [t\rangle\ v$ may take place,

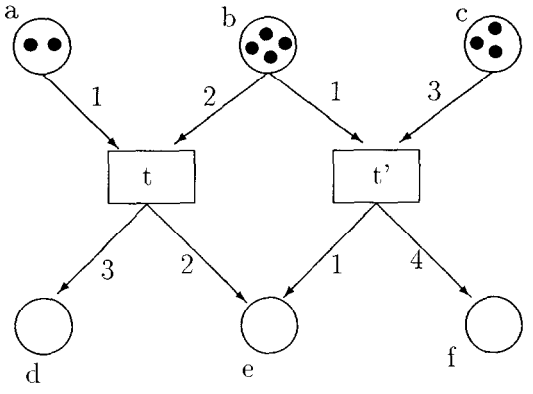

Fig. 1. A place/transition net.

with $v = (u -^\bullet t) \cup t^\bullet$. In general, a step may consist of the firing of a set of transitions, each disjointly enabled.

Let us consider the net in Fig. 1. It has a set of places $S = \{a, b, c, d, e, f\}$ and a set of transitions $T = \{t, t'\}$. Incoming and outgoing arrows of a transition and the associated numbers specify presets and postsets. For instance, ${}^\bullet t = \{a, 2b\}$ and $t^\bullet = \{3d, 2e\}$. The figure shows also the marking $u = \{2a, 4b, 3c\}$, namely the marking where there are two tokens on place a, four on place b and three on place c. The simultaneous firings of t and t' (namely the step $u\ [t, t'\rangle\ v$) produces the marking $v = \{a, b, 3d, 3e, 4f\}$.

Another example of a net is shown in Fig. 2(a). It has the characteristic to be 1-safe, i.e. such that each marking reachable from the initial one by a sequence of steps has only single occurrences of places.

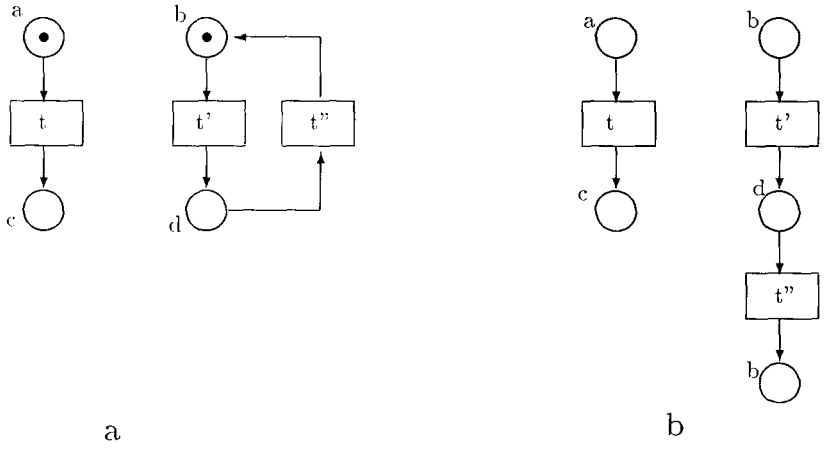

Fig. 2. A 1-safe net and a process.

Place/transition nets have an associated transition system, called *marking graph*, where the nodes are the markings of the net and the arcs are the steps.

In order to characterize the behavior of a net, *nonsequential processes* were defined in [4,21], corresponding to the unfolding of the net starting from some initial marking. They are defined as acyclic nets with places with single multiplicity, called *occurrence nets*, equipped with mappings to the original net. Actually, the processes defined in [4] are more abstract than those defined in [21], but we will not insist on this difference, because it is irrelevant for 1-safe nets to which we will refer in the paper. For example, a nonsequential process of the net in Fig. 2(a) is shown in Fig. 2(b). Note that for 1-safe nets each sequence of steps yields a process, while a process may represent different steps sequences; for example, sequences of steps corresponding to the process in Fig. 2(b) are:

$$\{a,b\}\ [t\rangle\ \{c,b\}\ [t'\rangle\ \{c,d\}\ [t''\rangle\ \{c,b\},$$
$$\{a,b\}\ [t,t'\rangle\ \{c,d\}\ [t''\rangle\ \{c,b\},$$
$$\{a,b\}\ [t'\rangle\ \{a,d\}\ [t\rangle\ \{c,d\}\ [t''\rangle\ \{c,b\}.$$

3.2. An algebraic approach to nets

In the paper we refer to an algebraic approach, proposed in [31] and further extended in [14], considering place/transition nets as graphs with a monoidal structure. Here we outline the results of these papers. The intuitive idea is to consider, given a net N, the markings as the free commutative monoid over the set of places, and a transition as an arc from the marking corresponding to its preset to the marking representing its postset. For example, in the net of Fig. 1 the markings form the free commutative monoid over $\{a,b,c,d,e,f\}$, t is a transition from $a \oplus 2b$ to $3d \oplus 2e$ (we use the notation $t : a \oplus 2b \to 3d \oplus 2e$) and t' is a transition from $b \oplus 3c$ to $e \oplus 4f$. Notice that the monoidal operation is denoted by $\oplus$. This operation captures the notion of parallelism between local states (places).

As a second step, the identity transition $id(u)$ is added for each marking u and a monoidal operation $\oplus$ is freely defined also on transitions, representing the concurrent firing of two of them. In this way a transition system $C[N]$ is defined, corresponding to the marking graph of N: the nodes are the markings and the transitions are the steps. The transition system $C[N]$ is defined by means of inference rules and axioms. An example of a rule is:

$$t : u \to v \text{ and } t' : u' \to v' \text{ implies } t \oplus t' : u \oplus u' \to v \oplus v'.$$

By this rule, the step consisting of the concurrent firing of t and t' in the net

above is represented by

$$t \oplus t' : 2a \oplus 4b \oplus 3c \to a \oplus b \oplus 3d \oplus 3e \oplus 4f.$$

The third step of the construction builds another structure $T[N]$ whose arcs are the computations of the net and, in particular, correspond to the nonsequential processes. This is obtained by further adding a sequentialization operation on the steps, denoted by ";", whose properties are defined by means of suitable inference rules and axioms. For instance, the inference rule

$$t : u \to v \text{ and } t' : v \to w \text{ implies } t;t' : u \to w$$

describes the sequentialization operator. But the fundamental axiom is the functoriality axiom

$$(t_1 \oplus t_2);(t_3 \oplus t_4) = (t_1;t_3) \oplus (t_2;t_4),$$

which expresses the intuitive concept that the parallel composition of two independent processes, each a sequentialization of two parts, has the same effect as a process where both first parts are executed in parallel, and then followed by both second parts. By this axiom two processes can be executed concurrently if and only if they can be executed in any order. For example, given the net in Fig. 2(a), the three terms

$$(t \oplus b);(c \oplus t'),$$
$$t \oplus t',$$
$$(a \oplus t');(t \oplus d),$$

corresponding to the sequences (without the last step) in the previous subsection, are equivalent by the functoriality axiom and the properties of the identities. For instance, we have that

$$(a \oplus t');(t \oplus d) = (a;t) \oplus (t';d)$$

by the functoriality axiom. Since a and d are identities, we have

$$a;t = t \quad \text{and} \quad t';d = t',$$

and as a consequence

$$(a \oplus t');(t \oplus d) = t \oplus t'.$$

In [31] all these structures are built by means of categorical constructions. Several categories of nets and net computations are defined, and the free constructions above correspond to adjoint functors between the categories.

3.3. An algebraic view of CCS

Process description languages, also called process algebras, are simple specification languages for concurrent systems, which permit to express the main concepts of concurrency. A calculus provides a set of constructors which can be seen as operations of an algebra. Specifications (or *agents*) written in these languages are thus terms of an algebra. This view of agents as terms simplifies the treatment of the language: for instance, the operational semantics can be given by induction on the terms, following the so-called SOS approach [39]. In this approach, a deductive system is associated to a calculus, with theorems of the form

$$p \xrightarrow{\mu} q$$

where p and q are terms and μ the label of the transition.

The resulting deductive system is a *natural deduction system*, where the inference rules have the following form:

$$\frac{p_1 \xrightarrow{\mu_1} q_1 \cdots p_n \xrightarrow{\mu_n} q_n}{p \xrightarrow{\mu} q}.$$

There is a transition from p to q if (and only if) the statement $p \xrightarrow{\mu} q$ is a theorem of the deductive system. Notice that a proof in such a system has the structure of a (inverted) tree, where the root of the tree is the theorem and the nodes are the premises used in the proof.

For instance, the rule corresponding to the choice of the left alternative in a nondeterministic choice in CCS is the following:

$$\frac{p \xrightarrow{\mu} q}{p + p' \xrightarrow{\mu} q},$$

where $+$ is the CCS operation denoting the nondeterministic composition of two processes.

Hence, a process algebra consists of an algebra (the agents) and of a set of transitions.

More recent works [8,17,34] present transitions themselves as terms of an algebra. The axioms are constants and a rule with n premises is an n-adic operation. A process algebra in this view is a two-sorted algebra, with one sort for states (the agents) and one for transitions. A ground term of the sort of transitions represents a proof in the SOS deductive system. Hence, the transitions and their proofs are identified. For instance, in the algebraic view of CCS, there is a proof constructor $<+\ p$ corresponding to the inference rule shown before. This constructor takes as argument a transition (which itself is

a term of the algebra) and is parameterized by an agent, which represents the rejected alternative (p' in the rule above).

Another proposal we know of to use an algebraic framework for the definition of concurrent systems is that of Astesiano and Reggio [1]. They aim at providing a general methodology for the algebraic specification of concurrent systems. Instead, we restrict ourselves to the algebraic structure underlying process description languages and Petri nets, offering specific axiomatizations and results.

3.4. CCS and Petri nets

In this subsection we recall some proposals for a truly concurrent operational semantics of CCS using Petri nets.

The approaches to the construction of net semantics for CCS that appear in the literature can be classified in two groups: those which consider nets as a semantic domain and define by structural induction a mapping which associates to each CCS term a net, and those which define in the SOS style [39] a suitable net for the whole language.

In the former group we can mention the works [20], where a mapping from CCS to occurrence nets and an operational semantics for CCS without restriction are given, and [43], where a denotational semantics for CCS based on event structures is presented. These approaches define implicitly or explicitly the operations of CCS over a subset of nets, and they have a denotational flavour. More recent approaches following this line can be found in [19,42].

Here, we are more interested in the latter approach. In these operational models compositionality is achieved directly (if the states are equipped with the operations of the language), and one can associate to each agent the net which is reachable from the initial marking corresponding to that agent. In this sense, the operational approach is more concrete and direct.

Some approaches force, either explicitly or implicitly, some transitions to be executed atomically. This line is followed in the SCONE approach [22,23] and in [12], where atomic actions are explicitly used.

In a similar way, an implicit use of atomic transitions is made in [13], where a distributed transition relation is introduced which does not define an actual Petri net.

In [10] a decomposition of each CCS agent in its sequential components is done and a condition/event system is defined by means of inference rules.

In [11] a condition/event system is defined, which plays the role of a distributed transition system for the whole CCS calculus. A relation called *decrel* is defined, which associates to each agent all the sets of components representing it.

In [36] a place/transition net is presented, and the characterization of markings associated to agents is done via the reachability relation of the net.

In [34] all the markings representing the same agent are made equivalent via an algebraic construction. However, the model is not a Petri net, precisely by the fact that some markings are made equal, and hence the model is the quotient of a net.

In [6] a truly concurrent semantics of CCS is given in terms of flow nets and flow event structures, and they are shown to coincide.

Although no formal proof of the coincidence of these models has been given, it is widely believed that they define the same truly concurrent semantics for CCS. Some proofs of correspondence have been given for some pairs of models; for instance for the flow net semantics of [6] and the permutation of transitions of [5], for the SCONE implementation [22] and the axiomatization of true concurrency for CCS of [17], for the event structure semantics of [43], and the net of [11].

We here follow the approach in [10–12,34], where, in order to give an actual distributed semantics, the sequential components of an agent are identified. In the case of CCS it is considered that the parallel constructor | is used to assemble sequential components. Hence, a term of the form $p|q$ (where p and q contain no parallel constructor) consists of two sequential components, p and q. However, it is important to keep track of how the components are assembled together. For instance, the CCS terms $(\alpha.NIL)\backslash\alpha|(\bar{\alpha}.NIL)\backslash\alpha$ and $(\alpha.NIL|\bar{\alpha}.NIL)\backslash\alpha$ are very different, and this distinction has to be reflected in their decompositions.

In [10–12,34], sequential components are called *grapes*, and the information about the context is simply recorded with two unary operations, called $|id$ and $id|$. A function called *dec* is defined, which, given a CCS term, returns its set of components. Function *dec* is defined by induction on terms, and reflects the intuition given above, for instance:

$$dec(p|q) = dec(p)|id \cup id|dec(q),$$

where the $|id$ and $id|$ operations are applied to all the elements of the sets $dec(p)$ and $dec(q)$.

For example,

$$dec((\alpha.NIL)\backslash\alpha|(\bar{\alpha}.NIL)\backslash\alpha) = \{(\alpha.NIL)\backslash\alpha|id, id|((\bar{\alpha}.NIL)\backslash\alpha)\},$$

$$dec((\alpha.NIL|\bar{\alpha}.NIL)\backslash\alpha) = \{(\alpha.NIL)|id\backslash\alpha, (id|(\bar{\alpha}.NIL))\backslash\alpha\}.$$

Each component is a place of the net, and transitions are defined by means of transition rules. However, the transition relation is rather complex, due mainly to the fact that markings (sets of grapes) which are not related by *dec* to any CCS term are reachable. The complication arises because in full CCS it is no more true that for each agent there is a unique set of grapes representing it. Instead, there is an infinite number of sets for each agent, each one representing a possible evolution of the system. In the calculus that we

present in this paper this problem does not arise, essentially because we forbid distributed choices.

The reader interested in the technical treatment of full CCS following our approach may refer to [44].

3.5. Equational type logic

The algebraic approach followed in the paper has the aim of unifying in a same framework different models. This has been made feasible by the use of equational type logic [29]. In this section we briefly recall this formalism. Typed algebras are classical one-sorted algebras enriched with a typing relation which is a binary relation, indicated by ":", on the elements of the algebra. Equational type logic provides a sound and complete language to reason about typed algebras. Basically, a presentation of a typed algebra is done by means of a set of conditional axioms of the kind

$$\frac{c_1, \ldots, c_n}{c}$$

where the logical conjunction of $c_1, \ldots, c_n$ is the condition under which c holds. Each of c_i and c may be either $t = t'$ (i.e. t is equivalent to t') or $t : t'$ (i.e. t is of type t'), where t and t' are terms. Note that the equation $t = t'$ induces as a *side-effect* a type assignment: in fact, if t has a different type from t', it acquires *also* the same type as t' by the axiom. As a consequence, a term may have more than one type. In [29] it is proved that any presentation has an initial model.

For readability reasons, in the paper we use grammars instead of typing axioms: each syntactic alternative represents a, possibly conditional, axiom. For example, if we have a signature with two constants, *NAT* and 0, and two unary operations, *succ* and *pred*, we write

Type *NAT*
$n ::= 0 | succ(n)$

instead of

$$\frac{0 : NAT \qquad n : NAT}{succ(n) : NAT}$$

Moreover, nonterminal symbols (e.g. n in the grammar above) will denote variables of the type. This means that, in any axiom, a variable t (possibly

with a subscript or a superscript) has to be considered as indicating a condition of the form $t : T$ *implies*. For example, the axiom

$$pred(succ(n)) = n$$

is equivalent to

$$\frac{n : NAT}{pred(succ(n)) = n}$$

Notice that, by the axioms, terms which were not explicitly typed by the grammar may obtain a type. For instance, the term $pred(succ(0))$, not typed by the above grammar, acquires type NAT, since it is equivalent to 0 and $0 : NAT$ is an axiom.

Equational type logic allows to deal with partiality (just giving a type only to elements which are defined) and polymorphism. For instance, the term $pred(0)$ is a term which has no type. In this way, *pred* is defined as a partial function from NAT to NAT. For our purposes the most interesting feature is that it is possible to present inference rules (by means of conditional axioms) and multiple typing, because the typing is a simple binary relation.

4. Semantics of a concurrent distributed language

4.1. SECCS and its interleaving semantics

In this paper we use a variation of CCS, called SECCS (for Simple, with parallelism Encapsulation, CCS), as our reference language.

The syntax is given in Table 2, where x is a variable;

$$\Delta = \{\alpha, \beta, \gamma, \ldots\}; \qquad \bar{\Delta} = \{\bar{\alpha} \mid \alpha \in \Delta\}; \qquad \tau \notin \Delta;$$

$\Delta \cup \bar{\Delta} \cup \{\tau\}$ is the set of basic actions, ranged over by μ; Φ is a permutation of $\Delta \cup \bar{\Delta} \cup \{\tau\}$ which preserves τ and the operation $^-$ of complementation; NIL represents an agent which cannot perform any action. Construct $\mu.p$ denotes an agent which can only perform action μ and then behaves like p. The actions of $p[\Phi]$ are renamings via Φ of those of p. Agent $p\backslash\alpha$ behaves like p but cannot perform actions α and $\bar{\alpha}$. Agent $p_1 + p_2$ can act either as p_1 or as p_2. Agent $p_1|p_2$ can perform in parallel the actions of p_1 and p_2; moreover agents p_1 and p_2 can synchronize, yielding τ, whenever they are able to perform complementary actions. The encapsulation operation is denoted by $\{\!|p|\!\}$. Agent $rec\,x.\ p$ denotes a recursive agent. Agents do not have occurrences of free variables.

The language SECCS is a simple version of CCS, where we impose some restrictions on terms:

- the recursion is guarded, that is, inside each occurrence of the recursive operator, any occurrence of the corresponding free variable is prefixed by an action. For example $rec\,x.\ \mu.x$ is guarded, while $rec\,x.\ \mu.x + x$ is not.
- the sum is guarded: in a summation context, each occurrence of a parallel operator is prefixed by an action or is inside an encapsulation operation. For example, $\mu.(\alpha.NIL|\beta.NIL) + \{\!|\alpha.NIL|\beta.NIL|\!\}$ is guarded, while $(\alpha.NIL|\beta.NIL) + \{\!|\alpha.NIL|\beta.NIL|\!\}$ is not.

The restriction to terms without free variables and to guarded recursion is usual in CCS. The newly imposed restriction—that of guarded sum—is not very strong from a practical point of view. In fact, terms of the form $p_1|p_2 + q_1|q_2$ are typically forbidden. This assumption is very natural if we consider that the distributed choice should not be primitive in the language.

The formalism that will be used throughout the exposition is that of *typed algebras* and *equational type logic* [29]. The model that we define is a typed algebra which subsumes the structure of

- the transition system of SECCS,
- the computations of the transition system,
- a Petri net for SECCS,
- its marking graph,
- the processes of the net.

By looking at certain subsets of types, particular structures can be observed, e.g. the transition system, or the net, or the net with its processes can be singled out. Table 1 gives the references to the tables of axioms presenting the different models, and may be useful for following the presentation.

Table 1
The axioms corresponding to SECCS models

Model	Axioms
Transition system	Table 2
Petri net	Tables 2 + 4
Marking graph	Tables 2 + 4 + 5
Processes	Tables 2 + 4 + 5 + 6
Computations	Tables 2 + 3
Relation between transition system and Petri net	Tables 2 + 4 + 5 + 6 + 3 + 7

We introduce in a completely modular way, step by step, the axioms that define each structure. The signature of the algebra is *fixed once and for all*: it has the operations and the types that appear in all presentations.

Table 2
SECCS and its transition system

Type AGENT

$p ::= NIL, x, \mu.p, p + p', p|p', p[\Phi], p\backslash\alpha, \{\!|p|\!\}, rec\, x.\ p$

rec $\quad p[rec\, x.\ p/x] = rec\, x.\ p$

act $\quad [\mu, p\rangle : \mu.p \xrightarrow{\mu} p$

res $\quad \dfrac{t : p_1 \xrightarrow{\mu} p_2, \quad \mu \notin \{\beta, \bar{\beta}\}}{t\backslash\beta : p_1\backslash\beta \xrightarrow{\mu} p_2\backslash\beta}$

rel $\quad \dfrac{t : p_1 \xrightarrow{\mu} p_2}{t[\Phi] : p_1[\Phi] \xrightarrow{\Phi(\mu)} p_2[\Phi]}$

sum $\quad \dfrac{t : p_1 \xrightarrow{\mu} p_2}{t <+\ p : p_1 + p \xrightarrow{\mu} p_2} \qquad \dfrac{t : p_1 \xrightarrow{\mu} p_2}{p\ +> t : p + p_1 \xrightarrow{\mu} p_2}$

enc $\quad \dfrac{t : p_1 \xrightarrow{\mu} p_2}{\{\!|t|\!\} : \{\!|p_1|\!\} \xrightarrow{\mu} \{\!|p_2|\!\}}$

par $\quad \dfrac{t : p_1 \xrightarrow{\mu} p_2}{t\rfloor p : p_1|p \xrightarrow{\mu} p_2|p} \qquad \dfrac{t : p_1 \xrightarrow{\mu} p_2}{p\lfloor t : p|p_1 \xrightarrow{\mu} p|p_2}$

syn $\quad \dfrac{t : p_1 \xrightarrow{\alpha} p_2, \quad t' : p_1' \xrightarrow{\bar{\alpha}} p_2'}{t|t' : p_1|p_1' \xrightarrow{\tau} p_2|p_2'}$

Initially, the typing relation is empty. The axioms define the typing by adding ⟨element, type⟩ pairs to the typing relation.

Also the conditions about free variables and guarded operations can be expressed by means of axioms and typing rules, but it is rather long and not particularly suggestive (see [44]).

Table 2 describes the language with the associated transition system defining the interleaving semantics. The elements of type AGENT are the SECCS terms. Note that we consider of type AGENT only those terms which are closed, guarded, and with guarded sum.

The moves of the transition system (type MOVE) are introduced by the axioms *act–syn*.

As a shorthand, we will use the notation

$$t : p \xrightarrow{\mu} p'$$

to establish that t is of type MOVE, that $source(t) = p$, $target(t) = p'$, and $label(t) = \mu$, where "*source*" is a function which gives the initial state of any MOVE; "*target*" is defined similarly, and "*label*" gives the action associated to the MOVE.

For each agent $\mu.p$ there is a MOVE $[\mu, p\rangle$ which source $\mu.p$ and target p. The other operations for MOVEs are proof constructors corresponding to the SOS rules: +> and <+ for the choices, ⌊ and ⌋ for the parallel merge, | for synchronization, $\backslash\alpha$ and $[\Phi]$ for restriction and relabelling respectively, and $\{\!|\}\!\}$ for encapsulation.

The axioms, except *enc*, correspond to the usual inference rules, in the SOS style, for the interleaving semantics of CCS. In this algebraic framework the rules are conditional axioms of the underlying algebra and each move of the transition system is identified with its proof, as presented in Section 3.

Notice that the agent $\{\!|p|\!\}$ behaves as p, since in the interleaving semantics parallelism is reduced to nondeterminism.

Recursion is handled by means of the *unfolding* axiom *rec* which permits to substitute a variable with its definition.

The precedence of the operators used in the presentation is given by the following list, ordered from less to more binding (the list contains also operators that will be defined in what follows):

$$/,\ rec\,x.\ ,\ ;,\ \oplus,\ +,\ <+\ ,\ +>,\ |,\ \lfloor,\ \rfloor,\ |id,\ id|,\ \mu.,\ [\Phi],\ \backslash\alpha,\ \%.$$

Parentheses are avoided whenever possible.

Example 4.1. The following SECCS term represents a system where two users share an exclusive resource. The resource can be requested and then released, and both users can perform an independent action when they do not possess the resource. The resource is represented by the following term.

$$p = rec\,x.\ \bar{\alpha}.\bar{\gamma}.x.$$

The users are represented by the terms

$$p_1 = rec\,x.\ \beta.x + \alpha.\gamma.x,$$
$$p_2 = rec\,x.\ \delta.x + \alpha.\gamma.x.$$

The whole system is represented by the term

$$sys = (p_1|(p|p_2))\backslash\alpha\backslash\gamma.$$

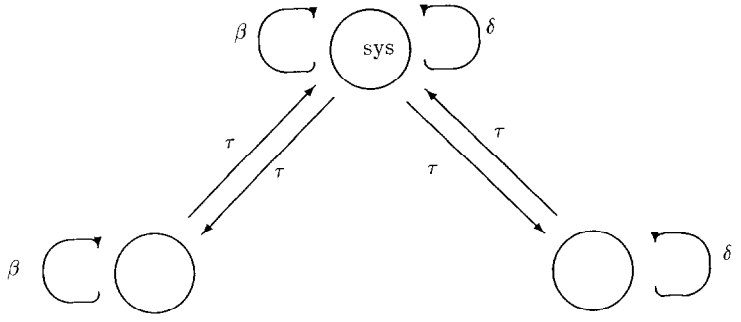

Fig. 3. The transition system corresponding to the term *sys*.

The term *sys* is an AGENT. Moreover, it can be proved, applying axiom *rec* to each component, that

$$sys = ((\beta.p_1 + \alpha.\gamma.p_1)|((\bar{\alpha}.\bar{\gamma}.p)|(\delta.p_2 + \alpha.\gamma.p_2)))\backslash\alpha\backslash\gamma.$$

From *sys* there are four possible MOVEs t such that $t : sys \xrightarrow{\mu} p$, two with label τ, one with label β, and one with label δ. The following term represents the MOVE with label β:

$$mv = (([\beta, p_1\rangle <+ \alpha.\gamma.p_1)\rfloor(p|p_2))\backslash\alpha\backslash\gamma : sys \xrightarrow{\beta} sys.$$

The complete transition system for *sys* is shown in Fig. 3.

4.2. *Computations of the transition system*

Given a transition system, the computations can be defined simply as sequential compositions of single moves. The unique important property of this concatenation is associativity. We will now include new axioms, shown in Table 3, to define COMPUTATIONs as sequential compositions of MOVEs.

The notation for COMPUTATIONs is $t : p \Rightarrow p'$, where p and p' are the source and target of t respectively.

Axiom *mtocom* retypes each MOVE as COMPUTATION. Axiom *seqcom* defines the sequentialization operation, and *assoc* states associativity.

Notice that COMPUTATIONs and MOVEs are elements of the same algebra, but some elements are COMPUTATIONs and not MOVEs.

Computations describe the behavior of a term. In fact, the tree of COMPUTATIONs ordered by prefix actually describes the complete behavior of an AGENT. However, this description is done from an interleaving point of view. Causality is not taken into account, and different interleavings of the

Table 3
Computations of the transition system

mtocom	$\dfrac{t : p \xrightarrow{\alpha} p'}{t : p \Rightarrow p'}$
seqcom	$\dfrac{t : p_1 \Rightarrow p_2, \quad t' : p_2 \Rightarrow p_3}{t;t' : p_1 \Rightarrow p_3}$
assoc	$\dfrac{t_1 : p_1 \Rightarrow p_2, \quad t_2 : p_2 \Rightarrow p_3, \quad t_3 : p_3 \Rightarrow p_4}{(t_1;t_2);t_3 = t_1;(t_2;t_3)}$

same "execution" (where only the order of occurrence of concurrent events is different) are considered as different computations.

Example 4.2.

$$c_1 = (([\beta, p_1\rangle <+ \alpha.\gamma.p_1)\rfloor(p|p_2))\backslash\alpha\backslash\gamma;$$
$$(p_1\lfloor(p\lfloor([\delta, p_2\rangle <+ \alpha.\gamma.p_2)))\backslash\alpha\backslash\gamma$$

is the computation from *sys* to *sys* where the first MOVE is *mv* of Example 4.1 and the following is the independent δ MOVE of the other user.

$$c_2 = (p_1\lfloor(p\lfloor([\delta, p_2\rangle <+ \alpha.\gamma.p_2)))\backslash\alpha\backslash\gamma;$$
$$(([\beta, p_1\rangle <+ \alpha.\gamma.p_1)\rfloor(p|p_2))\backslash\alpha\backslash\gamma$$

is a permutation of c_1. We have $c_1 \neq c_2$.

4.3. Truly concurrent semantics for SECCS

In this section we introduce a Petri net describing the truly concurrent semantics of SECCS in an algebraic way.

To define the net for SECCS we proceed by decomposing each SECCS agent in its set of sequential components, following the approach described in Section 3.4. In doing that, given an agent, a set of sequential components is returned as a result, typed PLACE. They constitute the places of the net. Each PLACE has some information about its context within the agent. This information is necessary to deal with restriction and nondeterminism. The syntax for PLACEs is given by the grammar of Table 4.

Operations $id|\_$ and $\_|id$ are unary operations which give the information about the parallelism context. For example, the PLACE $(id|\alpha.NIL)\backslash\beta$ represents a sequential component that can do an α action and stop. The term also

says that this component is in parallel with one or more PLACEs, and that the whole system is under the restriction of a $\_\backslash\beta$ operation.

Elements of type MARKING are the markings of the net: they are multisets of places. The empty multiset is denoted by 0.

Axioms *ACI* define the structure of a Petri net on the algebra, following the approach described in Section 3.2. In fact, markings with the operation $\oplus$ have the structure of a free commutative monoid over places. The constant 0 is the identity of the monoid.

Axioms *dec–rdispar* establish the correspondence between agents and markings of the net. They can be used as oriented rules to compute the set of PLACEs corresponding to a single agent. In fact, they define the function *dec* introduced in Section 3.4.

As pointed out in Section 3, a grammar does not define all the terms of a given type: for example, the left-hand side of the axiom *dec*, $m|m'$, which had no type according to the grammar, inherits the type MARKING from the right-hand side.

Among all markings, there are only some which are of interest: those which are equivalent to an AGENT, i.e. which can be typed also as AGENTs. For instance, the term $id|\alpha.NIL \oplus id|\beta.NIL$ is not of type AGENT. On the other hand, all AGENTs are represented by multisets of PLACEs; in fact, the following theorem holds.

Theorem 4.3. *Each AGENT is also a MARKING.*

Proof (*Outline*). Immediate by induction on AGENTs. □

Let us show, for example, that the AGENT $\alpha.NIL|\beta.NIL$ is also of type MARKING. We know that $\alpha.NIL$ and $\beta.NIL$ are of type PLACE (by the grammar), and the same holds for $\alpha.NIL|id$ and $id|\beta.NIL$. They are also MARKINGs since each PLACE is also a MARKING. Then, $\alpha.NIL|id \oplus id|\beta.NIL$ is also a MARKING. Thus, we can apply axiom *dec*, and $\alpha.NIL|\beta.NIL$ acquires type MARKING.

Moreover, it can be proved that each sequential AGENT (i.e. one not having a $\_|\_$ operator outside the occurrence of a $\mu.p$ or a $\{\!|p|\!\}$ operation) is also of type PLACE, i.e. it is represented by a single place in the net. For example, there is one PLACE for the AGENT $\alpha.(\beta.NIL|\gamma.NIL)$, but there are two PLACEs for $\beta.NIL|\gamma.NIL$, namely $\beta.NIL|id$ and $id|\gamma.NIL$. Hence, the occurrence of α represents a fork operation which has the side-effect of decomposing the AGENT $\beta.NIL|\gamma.NIL$ into its components.

Axioms *ACT–SYN* define the TRANSITIONs of the net. A notation similar to that of MOVEs is used for TRANSITIONs: the statement $t : m_1 \overset{\mu}{\longmapsto} m_2$ declares t as an element of type TRANSITION and defines its source, target, and label.

Table 4
A Petri net for SECCS

Type PLACE
$g ::= NIL, \mu.p, g[\Phi], g\backslash\alpha, g|id, id|g, g + g, \{\!|p|\!\}$

Type MARKING
$m ::= g, 0, m \oplus m$

ACI $\quad m_1 \oplus m_2 = m_2 \oplus m_1$
$(m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3)$
$m \oplus 0 = m$

dec $\quad m|m' = m|id \oplus id|m'$

disren $\quad (m_1 \oplus m_2)[\Phi] = m_1[\Phi] \oplus m_2[\Phi]$

disres $\quad (m_1 \oplus m_2)\backslash\alpha = m_1\backslash\alpha \oplus m_2\backslash\alpha$

ldispar $\quad (m_1 \oplus m_2)|id = m_1|id \oplus m_2|id$

rdispar $\quad id|(m_1 \oplus m_2) = id|m_1 \oplus id|m_2$

ACT $\quad [\mu, p\rangle : \mu.p \overset{\mu}{\longmapsto} p$

RES $\quad \dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2, \ \ \mu \notin \{\beta, \bar{\beta}\}}{t\backslash\beta : m_1\backslash\beta \overset{\mu}{\longmapsto} m_2\backslash\beta}$

REL $\quad \dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2}{t[\Phi] : m_1[\Phi] \overset{\Phi(\mu)}{\longmapsto} m_2[\Phi]}$

SUM $\quad \dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2}{t <\!\!+\ p : m_1 + p \overset{\mu}{\longmapsto} m_2} \qquad \dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2}{p\ +\!\!>\ t : p + m_1 \overset{\mu}{\longmapsto} m_2}$

ENC $\quad \dfrac{t : p_1 \overset{\mu}{\longrightarrow} p_2}{\{\!|t|\!\} : \{\!|p_1|\!\} \overset{\mu}{\longmapsto} \{\!|p_2|\!\}}$

PAR $\quad \dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2}{t|id : m_1|id \overset{\mu}{\longmapsto} m_2|id} \qquad \dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2}{id|t : id|m_1 \overset{\mu}{\longmapsto} id|m_2}$

SYN $\quad \dfrac{t : m_1 \overset{\alpha}{\longmapsto} m_2, \ \ t' : m_1' \overset{\bar{\alpha}}{\longmapsto} m_2'}{t|t' : m_1|m_1' \overset{\tau}{\longmapsto} m_2|m_2'}$

Notice that the effect of the $\{\!|\}$ construct is to reestablish the interleaving semantics. That is, the meaning of $\{\!|p|\!\}$ is "the agent p, but with its interleaving semantics". If p can do a MOVE t, then $\{\!|p|\!\}$ can do the TRANSITION $\{\!|t|\!\}$. For instance, the AGENT $\{\!|\beta.NIL|\gamma.NIL|\!\}$ is represented by a single PLACE in the net.

A term of type TRANSITION may also have the type MOVE, while there are terms which are only MOVEs (for example, $t\rfloor p$) and others which are only TRANSITIONs (for example, $t|id$). When a term has two types, it can be proved that the source, target, and label declared in both tables coincide.

Example 4.4. The AGENT *sys* of our running example is equal to the MARKING

$$p_1|id\backslash\alpha\backslash\gamma \oplus (id|(p|id))\backslash\alpha\backslash\gamma \oplus (id|(id|p_2))\backslash\alpha\backslash\gamma.$$

In fact

$$\begin{aligned} sys &= (p_1|id \oplus id|(p|p_2))\backslash\alpha\backslash\gamma && \text{by axiom } dec \\ &= (p_1|id \oplus id|(p|id \oplus id|p_2))\backslash\alpha\backslash\gamma && \text{by axiom } dec \\ &= p_1|id\backslash\alpha\backslash\gamma \oplus (id|(p|id))\backslash\alpha\backslash\gamma \oplus (id|(id|p_2))\backslash\alpha\backslash\gamma \\ & && \text{by axioms } rdispar \text{ and } disres \end{aligned}$$

Notice that each element of the sum corresponds to a sequential component of the system: the resource, and the two users.

The TRANSITION corresponding to the MOVE *mv* of Example 4.1 is represented by the term:

$$tr = ([\beta, p_1\rangle <+ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma : p_1|id\backslash\alpha\backslash\gamma \xrightarrow{\beta} p_1|id\backslash\alpha\backslash\gamma.$$

The complete net for *sys* is shown in Fig. 4. Places 1, 2, and 3 correspond to the components of *sys*. Places 4 and 5 correspond to the state of each user when the resource is assigned to her/him. Place 6 represents the state of the resource allocated to some user. Hence, markings $6 \oplus 4$ or $6 \oplus 5$ correspond to the global state where the resource has been assigned to a user.

Given a MARKING which is an AGENT, TRANSITIONs are very similar to MOVEs. The main difference is that TRANSITIONs take into account just the elements that are involved in the evolution. Instead, MOVEs are global. For instance *tr* in the example above describes only the active component of the system, which is on the left-hand side, and uses an $|id$ operator at the right-hand side, while *mv* contains the information that the right-hand side component of the system is $p|p_2$. Each TRANSITION may be completed in many ways and thus may correspond to several MOVEs.

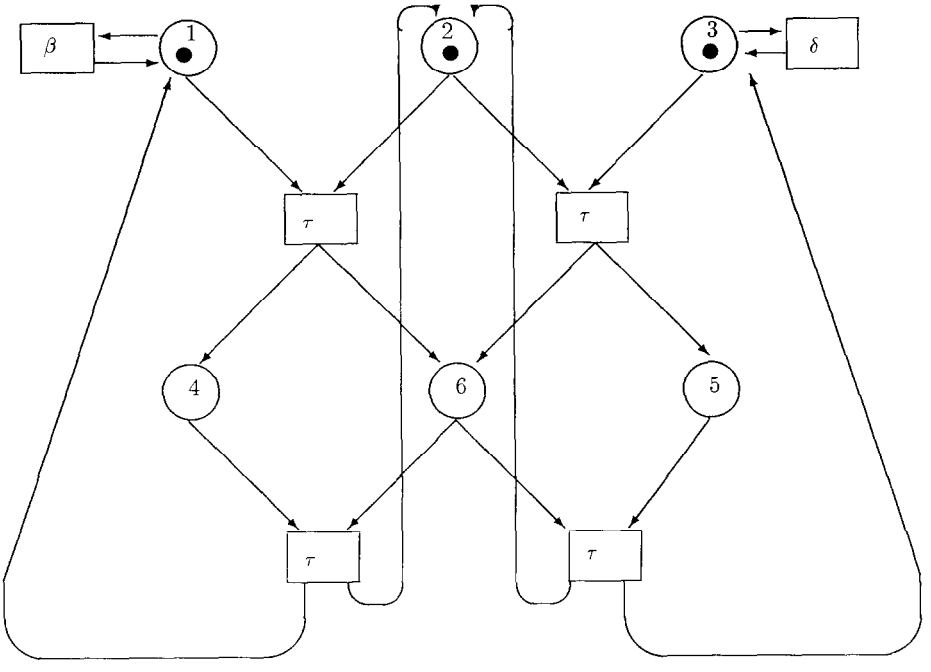

Fig. 4. The Petri net corresponding to the term *sys*.

4.4. Marking graph

The general construction from a net N to its marking graph $C[N]$, which has been described in Section 3.2, is here done by means of the axioms in Table 5.

A different type is given for the arcs of the marking graph. This type is called STEP, and, as a shorthand, we use

$$t : m_1 \overset{A}{\Longmapsto} m_2$$

to denote a STEP t with source m_1, target m_2, and a label A. Labels of STEPs are multisets of actions: STEPs correspond to the simultaneous occurrence of transitions.

For each marking m of the net axiom *idst* introduces a STEP called *identity* denoted by $m\%$. Identities have as source and target the same marking and empty label, and represent idle tokens in the net: tokens that in a certain tick of the clock stay idle without participating in any move. Then, all TRANSITIONs are re-typed as STEPs by axiom *ttost*, and the STEPs are closed by the $\_ \oplus \_$ operation which represents the parallel occurrence of events (axiom *parst*). The $\_ \oplus \_$ operation is required to satisfy the associativity, commutativity, and identity properties, that is, the structure of STEPs turns out to be a

Table 5
Construction of the marking graph of the net

idst	$m\% : m \overset{\emptyset}{\models\!\Rightarrow} m$
ttost	$\dfrac{t : m_1 \overset{\mu}{\longmapsto} m_2}{t : m_1 \overset{\{\mu\}}{\models\!\Rightarrow} m_2}$
parst	$\dfrac{t : m_1 \overset{A}{\models\!\Rightarrow},\ t' : m_1' \overset{A'}{\models\!\Rightarrow} m_2'}{t \oplus t' : m_1 \oplus m_1' \overset{A \uplus A'}{\models\!\Rightarrow} m_2 \oplus m_2'}$
iddis	$(m_1 \oplus m_2)\% = m_1\% \oplus m_2\%$
ACIst	$t \oplus t' = t' \oplus t, \quad (t \oplus t') \oplus t'' = t \oplus (t' \oplus t''), \quad t \oplus 0\% = t$ where t, t', and t'' are STEPs

free commutative monoid over TRANSITIONs (axiom *ACIst*). Moreover, identities distribute over the $\oplus$ operator (axiom *iddis*).

In this way the two different structures (the net and the graph) live together in the same algebra. Associating the MARKINGs with TRANSITIONs or with STEPs one can choose which structure one wants to see.

Example 4.5. The STEP corresponding to the MOVE *mv* of the system *sys* (see Example 4.1) is

$$([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma \oplus (id|(p|id))\backslash\alpha\backslash\gamma\% \oplus (id|(id|p_2))\backslash\alpha\backslash\gamma\%.$$

The two identities show that the second and third components remain idle while the first performs the action.

Figure 5 shows the marking graph corresponding to *sys*.

4.5. Nonsequential processes

Certain descriptions of behaviors are able to reflect all the information about causality and distribution. For place/transition Petri nets, nonsequential processes, described in Section 3, are such a description.

Table 6 presents the PROCESS type (denoted by $t : m_1 [\Rightarrow m_2$).

Axiom *stopr* re-types STEPs as PROCESSes. Axiom *parpr* defines the $\oplus$ operation on processes, while *Seq* introduces the concatenation operation.

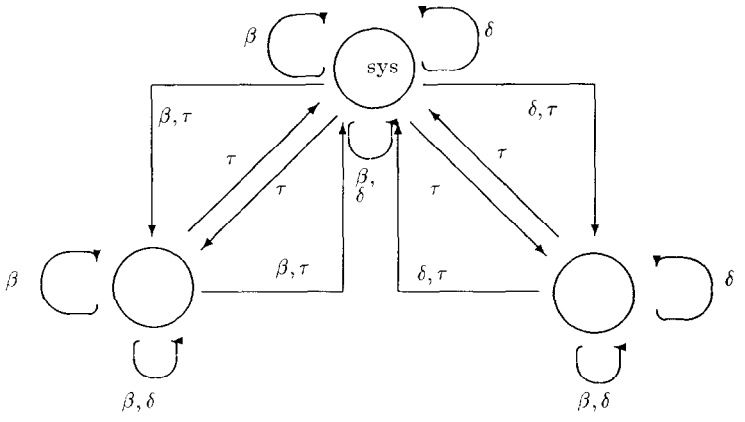

Fig. 5. The part of the marking graph of the net corresponding to the term *sys*.

Table 6
Processes of the net

stopr	$\dfrac{t : m_1 \overset{A}{\Longrightarrow} m_2}{t : m_1 [\Rightarrow m_2}$
parpr	$\dfrac{t : m_1 [\Rightarrow m_2,\ \ t' : m_1' [\Rightarrow m_2']}{t \oplus t' : m_1 \oplus m_1' [\Rightarrow m_2 \oplus m_2']}$
Seq	$\dfrac{t : m_1 [\Rightarrow m_2,\ \ t' : m_2 [\Rightarrow m_3}{t; t' : m_1 [\Rightarrow m_3}$
assoc	$\dfrac{t_1 : m_1 [\Rightarrow m_2,\ \ t_2 : m_2 [\Rightarrow m_3,\ \ t_3 : m_3 [\Rightarrow m_4}{(t_1; t_2); t_3 = t_1; (t_2; t_3)}$
func	$\dfrac{t_1 : m_1 [\Rightarrow m_2,\ \ t_2 : m_2 [\Rightarrow m_3,\ \ t_1' : m_1' [\Rightarrow m_2',\ \ t_2' : m_2' [\Rightarrow m_3'}{(t_1 \oplus t_1'); (t_2 \oplus t_2') = (t_1; t_2) \oplus (t_1'; t_2')}$
Id	$\dfrac{t_1 : m_1 [\Rightarrow m_2}{t_1; m_2\% = m_1\%; t_1 = t_1}$
ACIpr	$t_1 \oplus t_2 = t_2 \oplus t_1, \qquad (t_1 \oplus t_2) \oplus t_3 = t_1 \oplus (t_2 \oplus t_3), \qquad t_1 \oplus \emptyset\% = t_1$ where t_1, t_2, and t_3 are PROCESSes

Axiom *assoc* states associativity of sequentialization, and axiom *Id* defines identities with respect to sequentialization.

Axiom *func* is the "functoriality" axiom described in Section 3.2.

Axioms *ACIpr* induce associativity, commutativity, and identity of the parallel operator $\oplus$.

Example 4.6. A nonsequential process from the term *sys* can be seen in Fig. 6. Transitions of the process are instances of transitions of the net of Fig. 4. Places are numbered with the numbers of the places they are instances of. The following term is a PROCESS representing the process of the figure.

$$
\begin{aligned}
&([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma \oplus (id|(p|p_2))\backslash\alpha\backslash\gamma\%;\\
&(p_1|(p|id))\backslash\alpha\backslash\gamma\% \oplus (id|id|([\delta, p_2\rangle <+\ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma;\\
&((\beta.p_1 +> [\alpha, \gamma.p_1\rangle)|[\bar{\alpha}, \bar{\gamma}.p\rangle|id)\backslash\alpha\backslash\gamma \oplus (id|(id|p_2))\backslash\alpha\backslash\gamma\%;\\
&([\gamma, p_1\rangle|[\bar{\gamma}, p\rangle|id)\backslash\alpha\backslash\gamma \oplus id|(id|p_2)\backslash\alpha\backslash\gamma\%.
\end{aligned}
$$

The initial segment of the process, corresponding to the parallel execution of an action β and an action δ, is described by the following two terms of type PROCESS, which we show to be equal.

$$
\begin{aligned}
proc = {} & ([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma \oplus (id|(p|p_2))\backslash\alpha\backslash\gamma\%;\\
& (p_1|(p|id))\backslash\alpha\backslash\gamma\% \oplus (id|id|([\delta, p_2\rangle <+\ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma
\end{aligned}
$$

is a PROCESS by axioms *stopr* and *parpr*. Also *proc'* is a PROCESS:

$$
\begin{aligned}
proc' = {} & (p_1|(p|id))\backslash\alpha\backslash\gamma\% \oplus (id|id|([\delta, p_2\rangle <+\ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma;\\
& ([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma \oplus (id|(p|p_2))\backslash\alpha\backslash\gamma\%.
\end{aligned}
$$

By axiom *func*,

$$
\begin{aligned}
proc = {} & (([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma; (p_1|(p|id))\backslash\alpha\backslash\gamma\%)\\
& \oplus((id|(p|p_2))\backslash\alpha\backslash\gamma\%; (id|id|([\delta, p_2\rangle <+\ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma)\\
= {} & ([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma\\
& \oplus(id|id|([\delta, p_2\rangle <+\ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma.
\end{aligned}
$$

by *Id* and, similarly,

$$
\begin{aligned}
proc' = {} & ([\beta, p_1\rangle <+\ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma\\
& \oplus(id|id|([\delta, p_2\rangle <+\ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma.
\end{aligned}
$$

Thus $proc = proc'$.

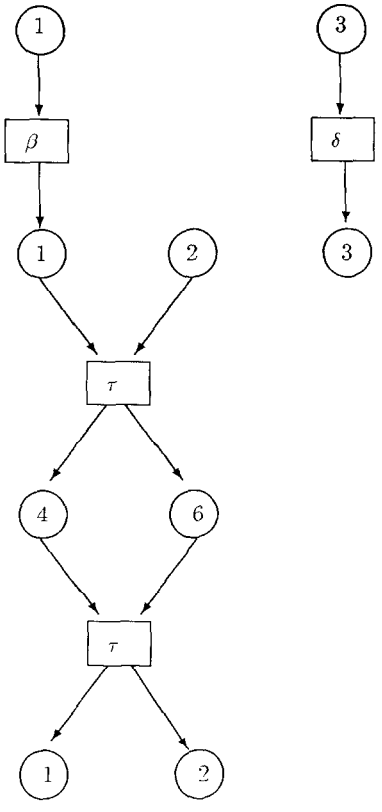

Fig. 6. A nonsequential process corresponding to the term *sys*.

4.6. Relating interleaving and truly concurrent semantics

The algebra we have presented above includes both the interleaving and the truly concurrent operational semantics. The different semantics are independent, i.e. they are described by different types, sharing the same set of states (AGENTs). For the interleaving semantics, we have the types MOVE and COMPUTATION. For the truly concurrent semantics, we have the types TRANSITION, STEP, and PROCESS.

Up to now, we introduced axioms defining non-interfering structures. The constructions defined independent models in a shared framework. Here we introduce some axioms relating the truly concurrent and interleaving semantics; we use the axioms in a different way than in the previous sections. Until now, each introduction of new axioms did not modify the previously defined structures. In other words, the congruence classes of previously typed terms did not change. For instance, when the axioms for PROCESSes where included in the algebra, MOVEs, TRANSITIONs, STEPs, and COMPUTATIONs remained

Table 7
Axioms relating the transition system and the marking graph

comid	$p\|id\% = p\%\|id$	$id\|p\% = (id\|p)\%$
	$p[\Phi]\% = p\%[\Phi]$	$p\backslash\alpha\% = p\%\backslash\alpha$
pardef	$\dfrac{t : p_1 \xrightarrow{\mu} p_2}{p\lfloor t = p\|id\% \oplus id\|t}$	$\dfrac{t : p_1 \xrightarrow{\mu} p_2}{t\rfloor p = t\|id \oplus (id\|p)\%}$
syncdef	$\dfrac{t : p_1 \xrightarrow{\mu} p_1',\ t' : p_2 \xrightarrow{\bar{\mu}} p_2'}{t\|(t' \oplus p\%) = (id\|p)\% \oplus t\|t'}$	$\dfrac{t : p_1 \xrightarrow{\mu} p_1',\ t' : p_2 \xrightarrow{\bar{\mu}} p_2'}{(t' \oplus p\%)\|t = p\|id\% \oplus t'\|t}$
pardis	$(t \oplus t')\backslash\alpha = t\backslash\alpha \oplus t'\backslash\alpha$	$(t \oplus t')[\Phi] = t[\Phi] \oplus t'[\Phi]$
	$(t \oplus t')\|id = t\|id \oplus t'\|id$	$id\|(t \oplus t') = id\|t \oplus id\|t'$
	where t and t' are STEPs	

unaltered, and it was possible to recover all structures by looking at the types of the elements.

Since here we want to explicitly relate different models, the axioms presented in this section modify the equivalence between previously typed terms, and in particular build larger congruence classes by unifying some of the previous ones, so leading to a coarser algebra. For instance, COMPUTATIONs which were different are identified. Hence it is no longer possible to recover the interleaving semantics.

The basic concept of the interleaving semantics is that of MOVE between states. Since TRANSITIONs are defined between local states and MOVEs between global ones, the concept corresponding to MOVE in the truly concurrent semantics is not TRANSITION but STEP. However, not all STEPs have a corresponding MOVE; the STEPs which correspond to MOVEs are those representing the occurrence of only one event.

Axioms *pardef* (see Table 7) decompose each parallel MOVE in the corresponding TRANSITION (for the only active component) and identities (for the rest of the system). Hence, it is similar to the *dec* axiom for states, since it permits to decompose a global MOVE in its local components. Axioms *syncdef* are similar to axioms *pardef* but for synchronizations (hence, the MOVEs have two active components).

Thus the transition system is included in the marking graph and it is only necessary to decompose some MOVEs and to let the $\_ \oplus \_$ operation distribute over SECCS operations (axiom *pardis*). The identity also distributes over SECCS operations (axiom *comid*). There is no need for the $\oplus$ and the %

operations to distribute over + or <+ , since in SECCS the sum is guarded. The following lemma holds.

Lemma 4.7. *Let t be a STEP. Then, using the axioms of Table 7, t|id and id|t are STEPs.*

Notice that for Lemma 4.7 axiom *comid* is necessary. In fact, we know from Table 5 that if m is a MARKING, $m\%$ is a STEP. Hence, to prove that $m\%|id$ is also a STEP, we use axiom *comid* to get $m\%|id = m|id\%$ and, since $m|id$ is a MARKING, $m|id\%$ is a STEP. For a similar reason, the rest of the *comid* axioms are introduced. Axioms *pardis* play a similar role.

The following theorem shows that the standard transition system for SECCS is included in the marking graph. We show the proof because it gives some insight on the axioms of Table 7.

Theorem 4.8. *Using the axioms of Table 7, each MOVE is also a STEP.*

Proof. The proof is done by induction on MOVEs.

For $[\mu, p\rangle$, we have, from axiom *ACT* in Table 4, that it has type TRANSITION. From axiom *ttost* in Table 5, it has also type STEP. In general, each MOVE which does not have an occurrence of a $\rfloor$, $\lfloor$, or | operation, is also a TRANSITION, and for them the inductive step is easy using the rules of Table 4. (Notice that the *ACT*, *RES*, *REL*, *SUM*, and *ENC* rules are the same as those for MOVEs.) The only interesting cases are $t\rfloor p$, $p\lfloor t$, and $t|t'$. We show the case of $t\rfloor p$.

By axiom *pardef* in Table 7, $t\rfloor p = t|id \oplus (id|p)\%$. Since p is an AGENT, we have from Theorem 4.3 that it is also a MARKING. The same holds for $id|p$. Hence, by axiom *idst* of Table 5, $(id|p)\%$ is a STEP. Since t is a STEP, we have by Lemma 4.7 that $t|id$ is also a STEP. Thus $t|id \oplus (id|p)\%$ is a STEP. □

As a consequence of Theorem 4.8 above, each COMPUTATION is a PROCESS. Thus, certain computations, where concurrent events are executed in different orders, are identified, because they correspond to equivalent PROCESSes. That means that causality is directly reflected in the model. Note that there are PROCESSes which have no representation as COMPUTATIONs. For instance, the PROCESS

$$([\beta, p_1\rangle <+ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma; ([\beta, p_1\rangle <+ \alpha.\gamma.p_1)|id\backslash\alpha\backslash\gamma$$

cannot be represented as a COMPUTATION.

Example 4.9. The two computations c_1 and c_2 of Example 4.2, which differ only in the order in which two independent events are executed, can be proved identical.

In this algebraic framework, the proof can be done in a totally formal way, and this provides a basis for automatic proof assistance.

$$c_1 = (([\beta, p_1\rangle <+ \alpha.\gamma.p_1)\rfloor(p|p_2))\backslash\alpha\backslash\gamma;$$
$$(p_1\lfloor(p\lfloor([\delta, p_2\rangle <+ \alpha.\gamma.p_2)))\backslash\alpha\backslash\gamma,$$

$$c_2 = (p_1\lfloor(p\lfloor([\delta, p_2\rangle <+ \alpha.\gamma.p_2)))\backslash\alpha\backslash\gamma;$$
$$(([\beta, p_1\rangle <+ \alpha.\gamma.p_1)\rfloor(p|p_2))\backslash\alpha\backslash\gamma.$$

By axioms *pardef*, *pardis*, and *dec*,

$$c_1 = (([\beta, p_1\rangle <+ \alpha.\gamma.p_1)|id)\backslash\alpha\backslash\gamma \oplus (id|(p|p_2))\backslash\alpha\backslash\gamma\%;$$
$$(p_1|(p|id))\backslash\alpha\backslash\gamma\% \oplus (id|id|([\delta, p_2\rangle <+ \alpha.\gamma.p_2))\backslash\alpha\backslash\gamma.$$

Thus, $c_1 = proc$, where *proc* is the term of Example 4.6. Similarly, $c_2 = proc'$, and by Example 4.6, $proc = proc'$. Hence, $c_1 = c_2$.

5. Designing distributed systems

In this section we discuss how our approach may help to face the different scenarios in the design of distributed systems.

Locality

A drawback of the transition system approach is that it describes transitions between global states only, and does not offer a full account of spatial distribution of control and of causal dependency among events taking place in independent/parallel subsystems. Thus the information on the structure of the system, present in the program, is lost. Instead, a Petri net maintains this information because its structure is intrinsically distributed.

In the working example shown in the previous section, all that we can see from the transition system is that it has three states, connected by some transitions. In the Petri net, however, the state is partitioned, and thus we can individuate the different parallel components of the system, i.e. the resource and the two users. This may be useful in order to derive a distributed implementation of the system which takes into account locality properties. Let us assume, for example, that instead of actions β and δ we had more complex processes, say p and q respectively. Since they do not communicate, we could locate them on different and distant processors.

Recoverability

In our approach it is possible to identify different sequential components of a system. For instance, in the process of Example 4.6, if a fault occurs after

the δ action, we can recover only the right-hand component since the left-hand one is not affected.

Another important characteristic of our approach is that, in the Petri nets we define, we have a syntactic counterpart (i.e. a term of our algebra) not only for the states, but also for the transitions. This allows us to concentrate on them in order to know where and between which components they occur. In the transition system of the example, we have two arcs labelled by action β, while in the Petri net there is only one transition β with a precise interpretation in terms of pre- and post-conditions. This helps recovery, because if an error occurs during the communication identified by β, we know which parts of the system must be examined.

Compositionality

Many formalisms for the specification of distributed systems are based on place/transition Petri nets. However, Petri nets lack compositional properties.

We associate to SECCS a net in a way similar to the SOS approach. Instead of rules we use conditional typed axioms, and hence we obtain an algebra which is a model for the language. The (distributed) semantics of the language is given by the subnet corresponding to each term. The algebraic structure of the language is inherited by the net: in the algebra the operations of the language can be applied to markings, giving markings as results. Thus we provide compositional constructors for the specification of systems.

Dynamic analysis

In our approach, a run of a system can be seen as a process of the net. A run of the system seen as a Petri process gives as much information as a large number of interleaved computations. For instance, the information given by the run of the process of Example 4.6 is the same as that given by many computations, among them those of Example 4.9. We emphasize that the number of interleaved computations corresponding to a given process may be exponential with the size of the process.

If the components of the system are loosely connected, i.e. they do not communicate very often, also the size of the transition system grows exponentially with the number of components, while this is not true for the Petri net. Consider the following agent, which is similar to that of the previous example, but where the two components requesting the resource make independently a sequence of actions:

$$\begin{aligned} sys1 = {} & ((rec\,x.\ \beta.\beta'.x + \alpha.\gamma.x) \mid \\ & (rec\,x.\ \bar{\alpha}.\bar{\gamma}.x) \mid \\ & (rec\,x.\ \delta.\delta'.x + \alpha.\gamma.x))\backslash\alpha\backslash\gamma. \end{aligned}$$

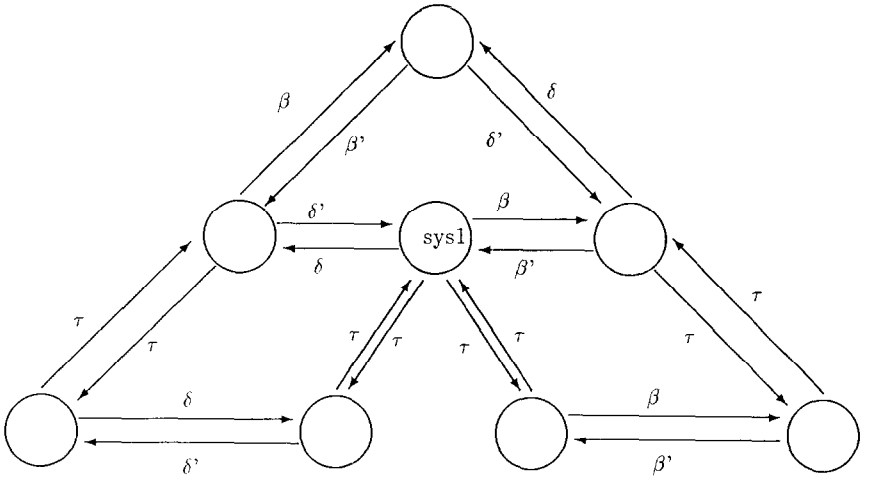

Fig. 7. The transition system for *sys1*.

Figure 7 shows the relative transition system and Fig. 8 the Petri net: while in the net there are only two new places and two new transitions, the transition system has five new states.

Configuration

SECCS permits to express the distributed nature of a system directly by the syntax. For instance, the term $\{\!|p_1|p_2|\!\}|p_3|p_4$ describes three sites, one of them containing two parallel components. Moreover, in the description of local components we keep track of the structure of the overall system by means of the $id|$ and $|id$ constructs.

Encapsulation

In our context, encapsulation is provided by the $\{\!|\,|\!\}$ construct. Let us consider, for example, the following program

$$p = \alpha.NIL|\beta.NIL|\gamma.NIL.$$

The corresponding Petri net is shown in Fig. 9(a), while Fig. 9(b) shows the net corresponding to

$$q = \{\!|\alpha.NIL|\beta.NIL|\!\}|\gamma.NIL,$$

where concurrency between the first two components is encapsulated. Note that we can choose between a sequential and a parallel behavior only if the two kinds of execution are different from a semantic point of view. This is true in our approach, but not in an interleaving one, where concurrency is not a primitive concept. In fact in the transition system model (see Table

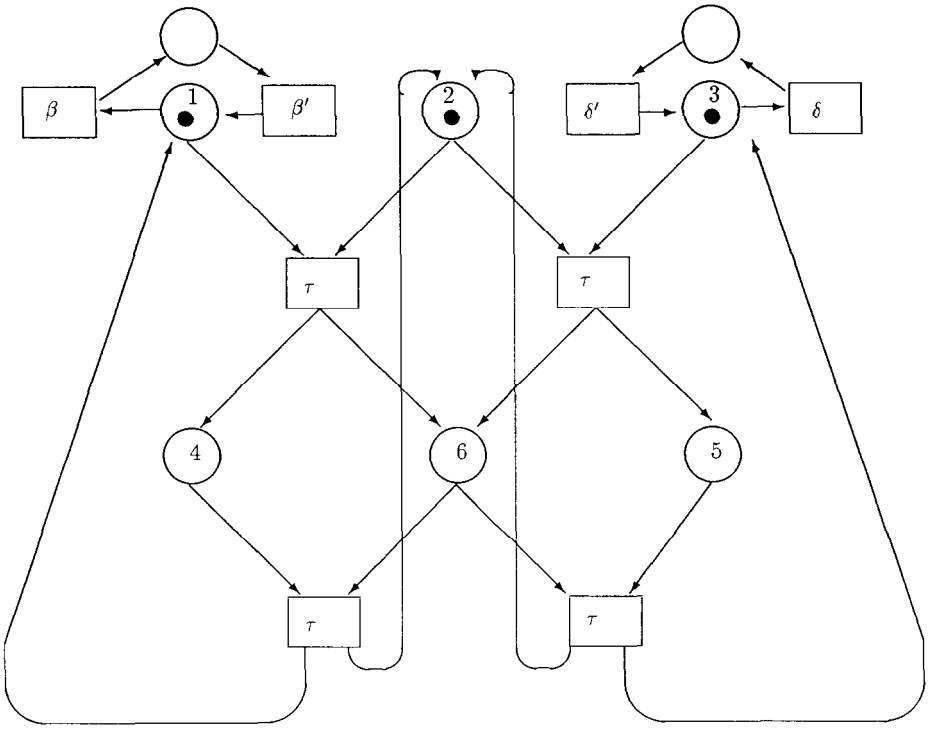

Fig. 8. The Petri net for *sys*1.

2), $\{\!|p|\!\}$ behaves exactly like p. The presence of constructs for controlling parallelism in existing languages and systems is a confirmation of the fact that programmers actually think in terms of true concurrency. Moreover they often need a specification closely related to the actual system on which programs execute. In this view, the choice of if and where to consider processes as executing either in an interleaving environment or in a fully concurrent way is an important option left to the designer.

The semantics of the construct is very naturally expressed in our model, where interleaving and true concurrency can be seen as two views of the system and are both included in the specification.

6. Further considerations

In this paper we have presented a specification language to describe distributed systems. The interleaving and truly concurrent semantics of the language have been given using an algebraic approach where the different structures of the two semantics are described in an unifying framework by different

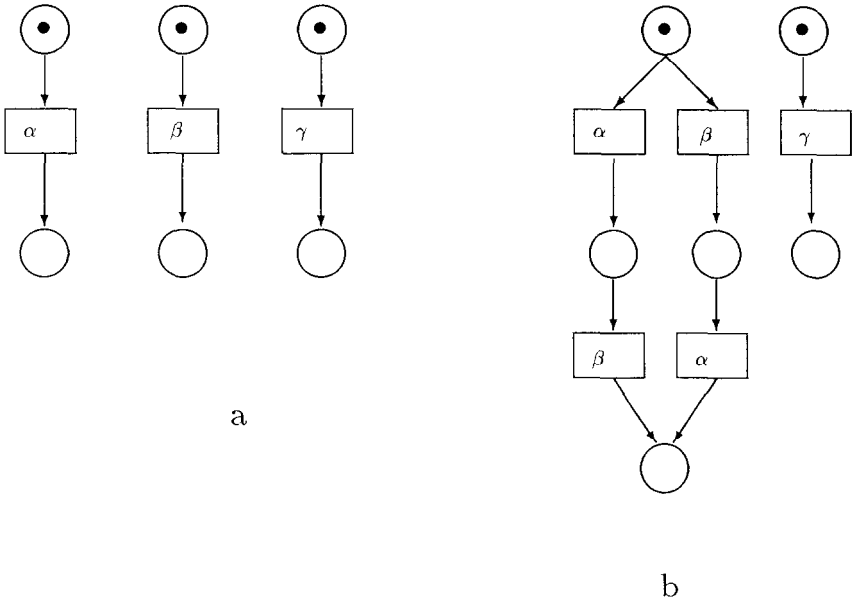

Fig. 9. Petri nets corresponding to $\alpha.NIL|\beta.NIL|\gamma.NIL$ and $\{|\alpha.NIL|\beta.NIL|\}|\gamma.NIL$.

equations and typings.

The approaches to the specification of distributed systems can be classified in two main groups: those based on a *shared resource* model, for instance [7,27], and those based on a *message passing* model. Our approach belongs to the latter group, together with the works on PDLs and Petri nets cited in the paper. Moreover, in this work we concentrate on the operational description of systems, and we do not consider any logical framework to express properties.

One of the most appealing features of our approach is that it can be seen not only as a formal model, but also as the basis on which an environment for the development of distributed systems and programs can be built. In fact, suitable tools could be implemented, based on the axioms of the algebra (including for example also graphical tools), which should be able to support the various aspects of design, debugging, and testing. The consistency among tools would be assured by the underlying model. The efficiency is probably acceptable since we do not need any theorem prover. In fact, despite of the axiomatic form, we can actually use interpreters for inference rules in the SOS style. Thus, we can efficiently execute SECCS terms in any of the models proposed just using the rules to determine moves, transitions, steps, or computations. Moreover, the axioms of Table 4 can be directed from left to right in order to efficiently decompose a SECCS agent in its sequential components.

We provide a framework in which control and data could be described in a homogeneous way, while in many models based on Petri nets and PDLs, data specification is introduced in an orthogonal manner. For example, in LOTOS [26], control is defined by a transition system and data are defined using algebraic specifications, and the language PCF is an extension of ACP with algebraic data specification [30]. In our context, actions themselves can be typed and introduced as elements of the same algebra, which may include both abstract data types and the description of control.

References

[1] E. Astesiano and G. Reggio, Direct semantics of concurrent languages in the SMoLCS approach, *IBM J. Res. Dev.* **31** (5) (1987).

[2] D. Austry and G. Boudol, Algèbre de processus et synchronisations, *Theoret. Comput. Sci.* **30** (1) (1984) 91–131.

[3] J. Bergstra and J. Klop, Algebra of communicating processes, in: J. de Bakker, M. Hazewinkel and J. Lenstra, eds., *Mathematics and Computer Science*, CWI Monographs **1** (North-Holland, Amsterdam, 1986) 89–138.

[4] E. Best and R. Devillers, Sequential and concurrent behavior in Petri net theory, *Theoret. Comput. Sci.* **55** (1) (1987) 87–136.

[5] G. Boudol and I. Castellani, A non-interleaving semantics for CCS based on proved transitions, *Fund. Inform.* **11** (4) (1988) 433–452.

[6] G. Boudol and I. Castellani, Three equivalent semantics for CCS, in: I. Guessarian, ed., *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science*, La Roche Posay, France, Lecture Notes in Computer Science **469** (Springer, Berlin, 1990) 96–141.

[7] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).

[8] A. Corradini, G. Ferrari and U. Montanari, Transition systems with algebraic structure as models of computations, in: I. Guessarian, ed., *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science*, La Roche Posay, France, Lecture Notes in Computer Science **469** (Springer, Berlin, 1990).

[9] J. de Bakker, W.-P. de Roever and G. Rozenberg, eds., *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Noordwijkerhout, Lecture Notes in Computer Science **354** (Springer, Berlin, 1989).

[10] P. Degano, R. De Nicola and U. Montanari, CCS is an (augmented) contact free C/E system, in: M.V. Zilli, ed., *Advanced School on Mathematical Models for the Semantics of Parallelism, 1986*, Lecture Notes in Computer Science **280** (Springer, Berlin, 1987) 144–165.

[11] P. Degano, R. De Nicola and U. Montanari, A distributed operational semantics for CCS based on condition/event systems, *Acta Inform.* **26** (1–2) (1988) 59–91.

[12] P. Degano, R. De Nicola and U. Montanari, Partial orderings descriptions and observations of nondeterministic concurrent processes, in: J. de Bakker, W.-P. de Roever and G. Rozenberg, eds., *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Noordwijkerhout, Lecture Notes in Computer Science **354** (Springer, Berlin, 1989) 438–466.

[13] P. Degano, R. De Nicola and U. Montanari, A partial ordering semantics for CCS, *Theoret. Comput. Sci.* **75** (1992) 223–262.

[14] P. Degano, J. Meseguer and U. Montanari, Axiomatizing net computations and processes, in: *Proceedings Fourth Annual Symposium on Logic in Computer Science*, Asilomar, CA 1989 175–185; also: *Inform. Comput.* (to appear).

[15] H. Ehrig and B. Mahr, *Fundamentals of algebraic specifications I*, EATCS Monographs on Theoretical Computer Science **6** (Springer, Berlin, 1985).
[16] J. Faust and H. Levy, The performance of an object-oriented threads package, *ACM SIGPLAN Notices, Proceedings OOPSLA 90* **25** (10) (1990) 278–288.
[17] G. Ferrari, Unifying models of concurrency, Ph.D. Thesis, Report TD-4/90, Dipartimento di Informatica, Università di Pisa (1990).
[18] C. Ghezzi, D. Mandrioli, S. Morasca and M. Pezzè, Symbolic execution of concurrent systems using Petri nets, *Comput. Language* **14** (4) (1989) 263–281.
[19] U. Goltz, On representing CCS programs by finite Petri nets, Arbeitspapiere der GMD 290, Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, Germany (1988).
[20] U. Goltz and A. Mycroft, On the relationship of CCS and Petri nets, in: J. Paredaens, ed., *Proceedings Eleventh ICALP*, Antwerp, Belgium, Lecture Notes in Computer Science **172** (Springer, Berlin, 1984) 196–208.
[21] U. Goltz and W. Reisig, The non-sequential behaviour of Petri nets, *Inform. Comput.* **57** (1983) 125–147.
[22] R. Gorrieri and U. Montanari, A simple calculus of nets, in: J. Baeten and J. Klop, eds., *Proceedings CONCUR 90*, Lecture Notes in Computer Science **458** (Springer, Berlin, 1990) 2–30.
[23] R. Gorrieri and U. Montanari, Distributed implementation of CCS, in: G. Rozenberg, ed., *Advances in Petri nets*, Lecture Notes in Computer Science (Springer, Berlin, to appear).
[24] I. Guessarian, ed., *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science*, La Roche Posay, France, Lecture Notes in Computer Science **469** (Springer, Berlin, 1990).
[25] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, NJ, 1985).
[26] ISO, Information processing systems—open systems interconnection—LOTOS—a formal description technique based on the temporal ordering of observational behaviour, ISO/TC97/SC21/N DIS8807 (1987).
[27] L. Lamport, A temporal logic of actions, Research Report 57, Digital Systems Research Center, Palo Alto, CA (1990).
[28] A. Maggiolo-Schettini and J. Winkowski, A compositional semantics for timed Petri nets, *Fund. Inform.* **XIII** (1990).
[29] V. Manca, A. Salibra and G. Scollo, Equational type logic, *Theoret. Comput. Sci.* **77** (1990) 131–159.
[30] S. Mauw and G. Veltink, A process specification formalism, *Fund. Inform.* **XIII** (1990) 85–139.
[31] J. Meseguer and U. Montanari, Petri nets are monoids: a new algebraic foundation for net theory, in: *Third Annual Symposium on Logic in Computer Science* (1988) 155–164.
[32] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980).
[33] R. Milner, *Communication and Concurrency* (Prentice-Hall, Englewood Cliffs, NJ, 1989).
[34] U. Montanari and D. Yankelevich, An algebraic view of interleaving and distributed operational semantics for CCS, in: *Proceedings Category Theory and Computer Science 89*, Lecture Notes in Computer Science **389** (Springer, Berlin, 1991) 5–20.
[35] M. Nielsen, CCS—and its relationship to net theory, in: W. Brauer, W. Reisig and G. Rozenberg, eds., *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri nets 1986, Part II; Proceedings of an Advanced Course*, Bad Honnef, Germany Lecture Notes in Computer Science **255** (Springer, Berlin, 1987) 393–415.
[36] E.-R. Olderog, Operational Petri net semantics for CCSP, in: G. Rozenberg, ed., *Advances in Petri nets* Lecture Notes in Computer Science **266** (Springer, Berlin, 1987) 196–223.
[37] J. Peterson, *Petri Net Theory and the Modeling of Systems*, (Prentice-Hall, Englewood Cliffs, NJ, 1981).
[38] G. Pinna, Petri nets and their composition problems, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa (1990).
[39] G. Plotkin, A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark (1981).

[40] W. Reisig, *Petri Nets: An Introduction*, EATCS Monographs on Theoretical Computer Science **4** (Springer, Berlin, 1985).

[41] W. Reisig, Petri nets and algebraic specifications, *Theoret. Comput. Sci.* **80** (1) (1991) 1–34.

[42] D. Taubner, *Finite Representation of CCS and TCSP Programs by Automata and Petri Nets*, Lecture Notes in Computer Science **369** (Springer, Berlin, 1989).

[43] G. Winskel, Event structure semantics for CCS and related languages, in: M. Nielsen and E. Schmidt, eds., *Proceedings Ninth ICALP*, Aarhus, Denmark, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 561–576; also: DAIMI Report PB-159, Computer Science Department, Aarhus University, Denmark (1983).

[44] D. Yankelevich, Parametric views of process description languages, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa (1992).